格式化字符串漏洞自动检测与测试用例生成 *

黄　钊，黄曙光，邓兆琨，黄　晖

(国防科技大学 电子对抗学院, 合肥 230037)

摘　要：格式化字符串漏洞是一种危害高、影响广的软件漏洞。当前漏洞检测方式存在人工依赖度高，误报率高，检测模型单一，未能充分考虑格式化字符串漏洞特点等多种局限性。针对以上问题，对格式化字符串漏洞特征进行分析，设计并实现了一种基于符号执行的格式化字符串漏洞自动检测与测试用例生成的系统。该方法可自动检测 Linux 下二进制程序中格式化字符串漏洞的存在性，判定其是否可能导致任意内存读写危害，并生成稳定有效的测试用例。

关键词：格式化字符串漏洞；符号执行；自动检测；测试用例生成

中图分类号：TP311　　doi: 10.3969/j.issn.1001-3695.2018.01.0168

Automatic detection and test cases generation of format string vulnerability based on symbol execution

Huang Zhao, Huang Shuguang, Deng Zhaokun, Huang Hui

(*Collage of Electronic Countermeasure*, *National University of Defense Technology*, *Hefei* 230037, *China*)

Abstract: The format string vulnerability is a kind of software vulnerability which has high risk and wide impact. Currently, there are many limitations of vulnerability detection method, such as high degree of artificial dependence, high false positive rate, single detection model and failing to consider the characteristics of the format string vulnerability fully. To solve above problems, this paper analyzed the format string vulnerability. Then based on symbolic execution, the paper designed and produced a way to detect formatted string vulnerability and generate test cases automatically. This method could detect the existence of the format string vulnerability in Linux binary program automatically. Then it determined whether it could lead to harm, which allowed attackers to read or write arbitrary memory. Meanwhile it generated stable and effective test cases.

Key words: format string vulnerability; symbolic execution; automatic detection; test cases generation

0　引言

格式化字符串漏洞在通用漏洞类型库 CWE 中解释为软件使用了格式化字符串作为参数，且该格式化字符串来自外部输入。其主要成因是格式化字符串函数对格式化控制符的解析缺陷及程序员对用户输入的不严格过滤，导致外部输入能够传递给格式化控制符并被解析[1]。

现有漏洞挖掘技术所发掘的程序错误并不一定能造成实质性危害，往往具有较高误报率和漏报率。因此针对漏洞特性，产生准确度高的稳定测试用例十分重要。现阶段漏洞检测方式上，人工漏洞检测技术效率低下且对人工经验依赖性强，往往需辅以自动化工具进行；传统漏洞自动检测工具，如 fuzzing 等，针对性不强、盲目生成测试用例导致误报率较高且路径覆盖率有限、穿透性不强。

符号执行[2]使用符号化输入来代替实际输入，该符号值可表示程序在此接收的所有可能输入，进而程序中的操作被转换为符号表达式的操作，条件判断也转换为符号约束。接着利用 Z3[3]、Yices[4]、STP[5]等约束求解器进行约束求解，即可在变量域中找到与各变量间均满足约束的特定变量值。利用符号执行和约束求解技术检测漏洞并生成测试用例的方法针对性强，对复杂代码逻辑穿透性好，生成的测试用例准确性高，在发现软件漏洞和生成测试用例方面有着较好效果。因此目前出现了一批基于符号执行自动化进行程序分析和漏洞检测的工具，如 Cha 等人提出的 Mayhem[6]、Huang 等人[7]提出的 CRAX 和 Shellphish 提出的 angr[8]等，但这些工具也存在漏洞检测模型较为单一等局限性，且未能充分考虑格式化字符串漏洞特点。以 CRAX 为例，其在漏洞存在性判定时，仅监视 IP 寄存器并以 IP 内容是否被外部输入符号化为判定标准，虽然该判定思路基本符合多数控制流劫持漏洞触发后的特征，但部分漏洞危害并非局限于 IP 符号化，如格式化字符串漏洞就可造成任意地址读写的危害。

本文针对现阶段格式化字符串漏洞检测技术中存在的缺陷，

收稿日期：2018-01-23；**修回日期：**2018-04-10　　**基金项目：**国家重点研发计划重点专项项目（2017YFB0802905）

作者简介：黄钊（1994-），女，江西吉水人，硕士研究生，主要研究方向为信息安全（hz0_mu@163.Com）；黄曙光（1964-），男，教授，博导，主要研究方向为信息安全；邓兆琨（1993-），男，硕士研究生，主要研究方向为信息安全；黄晖（1987-），男，博士，主要研究方向为信息安全、程序验证.

以准确检测并生成相应测试用例为目标，针对 linux 系统下由 C 语言或 C++编写生成的小规模二进制程序，设计了一个基于符号执行的格式化字符串漏洞自动检测与测试用例生成方法，并基于 S2E 符号执行引擎实现了一个格式化字符串漏洞检测与测试用例生成系统 FSVDTG（format string vulnerability detection and test cases generation）。

1　格式化字符串漏洞

1.1　格式化字符串函数与格式化控制符

格式化字符串函数即向标准输出设备或文件等指定的地方输出指定格式内容的函数，ANSI C 规范中定义了大量格式化字符串函数。典型格式化字符串函数如表 1 所示。

表 1　典型格式化字符串函数

函数	作用
printf	打印到 stdout 流
fprintf	打印到 FILE 流
sprintf	打印到字符串
snprintf	打印一定长度字符到字符串
vprintf	从 va_arg 结构打印到 stdout 流
vfprintf	从 va_arg 结构打印到 FILE 流
vsprintf	从 va_arg 结构打印到字符串
vsnprintf	从 va_arg 结构打印一定长度字符到字符串

除了上述标准函数外，还有用于输出到 syslog 设施的 syslog 函数、用于设置 argv[]的 setproctitle 函数，与其他类似 err*、verr*、 warn*、 vwarn*的函数等。

表 2　部分格式化控制符

格式化控制符	含义	对应参数值
%x	以十六进制形式格式化	值
%s	以字符串形式格式化	指针（引用）
%n	将已打印字符数(DWORD 值）输出到指定变元所指向地址	指针（引用）

函数参数方面，以最常见的 printf 为例，其参数主要包括格式化控制符和待输出数据列表两部分。其中格式化控制符用于指定待输出数据的输出格式，其后可填充待输出变量参数，格式化控制符与变量参数对应[9]。格式化控制符存在许多不同的数据类型，本文主要介绍与格式化字符串漏洞紧密相关的格式化控制符，如表 2 所示。

1.2　栈与格式化控制符

格式化字符串函数根据格式化控制符的指示从栈空间取得参数，如对于如下 C 语句，栈空间的排列分布如图 1 所示。

```
printf ("a = %d, b = %d, c = %08x\n", a, b, &c);
```

正常情况下格式化控制符中的每一个百分号“%”对应栈空间中一个参数，但由于格式化字符串函数是可变参函数，即使百分号“%”的数目与格式化字符串函数实际参数数目不相同，格式化字符串函数也无法察觉，并仍会继续从栈空间的下一个区域取出对应数据作为下一个参数，并按照给定格式化控制符进行解析。

除上述例子展示的格式化字符串漏洞可造成的内存信息泄露问题外，通过“%n”格式化控制符还可进行内存数据修改[10]，甚至覆盖函数返回地址、GOT 表地址等，达到控制流劫持的目的[11,12]。

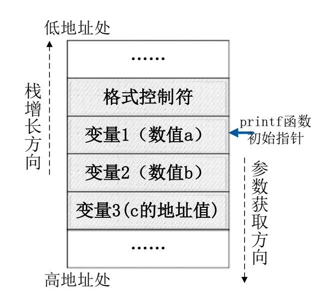


图 1　正常运行时 printf 函数栈空间分布

2　格式化字符串漏洞检测与测试用例生成

2.1　系统整体架构

通过对格式化字符串漏洞的分析，为准确检测并有针对性的生成稳定的测试用例，本文基于格式化字符串漏洞原理和符号执行与约束求解技术思想，在 S2E 符号执行引擎的基础上，设计并实现了格式化字符串自动检测与测试用例自动生成系统 FSVDTG。

FSVDTG 系统首先定义 printf、sprintf 等格式化字符串函数为不安全函数，并将其中的格式化控制符参数记为 format 参数，同时定义格式化字符串函数中的 format 参数内容不应是符号值的安全策略。然后挂钩不安全函数，收集到达不安全函数程序点的路径上的约束形成路径约束，并在到达不安全函数程序点时进行安全策略检测。若符合安全策略，则继续执行；若违反安全策略，则在约束集合中添加可触发格式化字符串漏洞危害的若干约束，由此构建测试用例约束，再在当前路径约束的前提下对该测试用例约束进行求解。若有解则认为存在程序漏洞并自动构造出可行解，即得到测试用例。后续执行测试并输入该测试用例时，其将能触发格式化字符串漏洞执行任意读写功能。系统运行流程如图 2 所示。

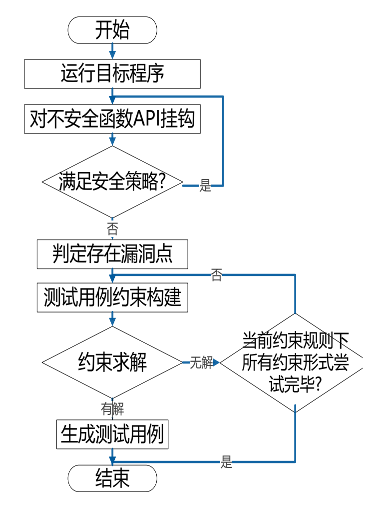


图 2　FSVDTG 系统运行流程

测试用例生成方面，格式字符串漏洞一般用于进行内存地址的读写，具体来说通常使用%x、%s 等格式化控制符进行任意地址内存读，使用%n 进行任意内存写，配合格式化控制符中的“$”即可构造一个简洁的测试用例以实现任意内存读写。例如“%3$n”，其中“3$”表示该“%n”对应于待输出数据列表中第 3 个参数指向的内存值，若格式化字符串函数的参数列表中没有第 3 个参数，则会写入从栈空间相应内存地址中。由此思路，本文可结合符号执行技术构建相应测试用例。

2.2 漏洞存在性检测

在格式化字符串漏洞的检测中，FSVDTG 系统首先应检测不安全函数调用处是否能满足既定安全策略，即检测格式化字符串函数的格式化字符串参数部分是否为符号值。若为符号值，则代表该 format 参数可被外部输入影响，进而认为可能存在可利用的漏洞。

具体实现上，首先对 printf 等格式化字符串函数进行挂钩。挂钩时提供静态挂钩和动态挂钩两种支持。静态挂钩时通过函数签名识别函数以确定函数静态地址，进而可由函数静态地址对函数进行挂钩；动态挂钩时通过自定义 init_env 库，并通过 LD_PRELOAD 加载 init_env 共享库，使得程序运行前优先加载自定义的 init_env 链接库来挂钩函数。然后通过回调函数获取其 format 参数在栈空间中的地址，并检查其内容是否为符号值。若为符号值，说明格式化字符串可被外部输入影响，则接下来转入构造相应测试用例约束；若不为符号值，则不作处理。

为确定 format 是否为符号值，搜索并记录下符号化区域的起始地址及长度，检测 format 地址是否在该符号化区域内。若在即可判断 format 参数是符号值，并将 format 参数所在的地址记为 format ，将 format 参数所在的符号区域起始地址记为 formatArea_StartAddr。

2.3 任意地址读测试用例约束构建

对于任意地址读，由于格式化漏洞产生时其 format 参数的内容来自于输入，且该输入内容往往也存在于栈空间，所以也可控制在该输入内容的起始处设置为准备读取的地址，然后配合“$”使得“%s”指向该地址值在栈上的位置，进而可通过字符串形式读取该地址处内容。

由此可构造形如“junk + addr + %B$s”的测试用例。其中 B 为某个正整数，使得通过“B$”能指向到的参数在栈上取得的对应恰是本文的 addr；addr 为本文准备读取的地址，可通过 lua 脚本自行指定。由于发生格式化串漏洞时输入部分会被存放在栈中，且成为格式化字符串函数的 format 参数，所以 addr 作为 format 参数的一部分也会存在于栈中的某个位置。若通过“B$”指向 addr，则就能使得“%s”从本文指定的 addr 开始读取直到\x00；而另一方面“$”选取参数时以 4 字节为单位，因此 junk 是作为凑齐 4 字节对其的填充字符。

表 3 约束构建过程函数/数组定义

函数名/数组名	含义
NotExpr	用于生成一元“取反”形式的 bool 型约束的函数
EqExpr	用于生成二元比较 “等于”形式的约束的函数
Memory	以内存地址为下标，记录该地址中内存值的数组

由此思路可构建相应利用约束，将 junk_constraint、addr_constraint 和 form_constraint 三项约束经过合取即形成测试用例约束 TestCase_ constraint，即如式（1）所示。

$$TestCase_constraint=junk_constraint \wedge addr_constraint \wedge form_constraint \tag{1}$$

为便于描述约束构建过程，定义如表 3 表示形式。

2.3.1 junk_constraint 约束

junk_constraint 约束即对 junk 字符部分的约束。由于格式化字符串发生时输入部分往往会读入缓冲区中进而存储在栈空间中，所以可将准备读取的地址放置在输入部分的开头。具体的 junk 字符数目计算如下：

```
junk_num = 4-(format-formatArea_StartAddr)%4;
if junk_num==4
      junk_num=0;
```

同时，为使得该段 junk 字符不影响 printf 参数的读入，该段字符不能包含“\x00”，则 junk_constraint 约束构建过程如下伪代码：

```
for i<- formatArea_StartAddr   to
formatArea_StartAddr+junk_num   do
      junk_constraint = NotExpr( EqExpr (Memory[i] , 0x00) )
```

2.3.2 addr_constraint 约束

addr_constraint 构建即对测试用例中 addr 区域是否能等于本文指定要读/写的地址值的约束。约束构建过程如下伪代码：

```
for i<-1 to 4 do
  addr_constraint = EqExpr(
  Memory[formatArea_StartAddr+junk_num+i] ,
  addr[i]   )
```

2.3.3 form_constraint 约束

form_constraint 约束即对测试用例中真正造成任意内存读写的格式化字符串部分的约束。

对于构造任意地址测试用例时，该部分形如“%B$s”，其中正整数 B 通过“$”符号保证此时格式化字符串函数要操作的参数指向准备写入的地址。其计算如下：

```
B= (format-formatArea_StartAddr-1)/4 +1;
```

由此将“%B$s”内容转换成字符串形式存储在 form_str 数组中，则 form_constraint 约束构建过程如下伪代码：

```
for i<-1   to len(form_str) do
addr_constraint =
EqExpr (Memory[formatArea_StartAddr+junk_num+4+i] , form_str[i] )
```

2.4 任意地址写测试用例约束构建

对于任意地址写，可构造形如“junk + addr + %Ax%B$n”或“junk + addr + %Cx%D$hn%Ex%F$hn”的测试用例，其中 junk、addr 与任意地址读时测试用例中的含义相同； A、C、D 为正

整数，用于控制当前已输出字符数；正整数 B、D、F 使得通过%n 对应参数为 addr；预写入的数据同样可通过 lua 脚本自行指定。

此时测试用例约束 TestCase_ constraint 仍由 junk_constraint、addr_constraint 和 form_constraint 合取形成，形如式（1），其中任意地址写时的 junk_constraint 及 addr_constraint 约束的构建均与任意地址读时类似，只需将准备进行读取的地址变换为准备进行写入的地址即可，故此处不再赘述，下文主要叙述任意地址写时的 form_constraint 约束构建。

form_constraint 约束是对“%Ax%B$n”或“%Cx%D$hn%Ex%F$hn”部分的约束，利用%n 可将已打印出的字符串长度写入内存的特性完成任意地址写入。具体来说，%n 能够一次性改写起始于该内存地址后一个 DWORD 长度的内容。若待写入数据过长，直接写入一整个 DWORD 长度可能需占用过长缓冲区，进而可能导致程序崩溃或等候时间过长，此时可通过%hn 来适时调整。%hn 功能与%n 类似，但%hn 一次性只改写一个 WORD，可通过 2 个%hn 分两次来改写一个 DWORD 长度内容。虽然使用%hn 时需要的已输出字符数较少，但由于需要进行多次写入，所得的测试用例就会变长，而部分环境下对输入缓冲区长度有所限制，导致所构造测试用例失效。因此%n 和%hn 应视情况调整使用。

表 4　form 部分测试用例形式

格式化控制符	使用条件	form 部分测试用例形式
%n	data<0x10000	%Ax%B$n
%hn	data≥0x10000	%Cx%D$hn%Ex%F$hn

具体构造“%Ax%B$n”形式的测试用例主体部分时，同样以将预写入的数据记为 data，且以 high 和 low 分别表 data 的两高位字节和两低位字节的数值，预进行写入的地址记为 addr。具体构造 form 部分测试用例形式如表 4 所示，其中 A、B、C、D、E、F 均为正整数。

其中 A、B 值计算如下：

```
A = data – junk – 4 ; //4 is the length of addr
B = ( format - formatArea_StartAdd -1 )/4    + 1;
```

而对 C、D、E、F，由于%n 和%hn 是根据当前格式化字符串函数已输出字符数进行数据写入的，故需比较 high 与 low 部分的数值大小，先写入其中的较小值，再写入较大值。具体计算如表 5 所示。

表 5　数值计算

计算条件	C	D	E	F
high < low	addr+2	high –junk -4	addr	low – high
high ≥ low	addr	low – junk -4	addr+2	high – low

2.5　约束求解及再次构建

通过将 junk_constraint、addr_constraint 和 form_constraint 三项约束进行合取即形成 TestCase 约束，然后调用约束求解器，在当前路径约束的前提下对该约束进行求解，使得所得解既能保证到达漏洞触发点，又能满足测试用例约束。若有解则可利用，再优化路径约束，然后进行求解，解出一个满足约束条件的实例；若无解，则增加一些 junk 字符后再行构建测试用例。由于“$”所指向部分以 4 字节为跨度，所以为使得增加 junk 字符后“$”仍指向 addr，每次重新构造约束时需进行如下调整：

```
junk_num = junk_num+4;
A = A-4;
B = B+1;
```

然后由此再按照 4. 3 节中步骤重新构建 TestCase 约束并求解，直到求得一个可行解或所构建的测试用例总体长度超出当前符号区域长度时停止。若得到可行解，将其输出即可得到完成任意地址读或任意地址写的测试用例。

3　实验

为直观说明实验效果，本文以 pingme 二进制程序的测试用例生成过程为例，对本文方法进行演示。该程序在 ida 中查看其伪代码情况如图 3 所示。其中 sub_804858B 函数伪代码如图 4 所示。

```
1  void __cdecl __noreturn main()
2  {
3    char format; // [sp+Ch] [bp-4Ch]@2
4    int v1; // [sp+4Ch] [bp-Ch]@1
5
6    v1 = *MK_FP(__GS__, 20);
7    sub_80485D7();
8    puts("Ping me");
9    while ( 1 )
10   {
11     if ( sub_804858B(&format, 64) )
12     {
13       printf(&format);
14       putchar(10);
15     }
16     else
17     {
18       puts(";( ");
19     }
20   }
21 }
```

图 3　pingme 程序伪代码

```
size_t __cdecl sub_804858B(char *s, int n)
{
  char *v3; // [sp+Ch] [bp-Ch]@1

  fgets(s, n, stdin);
  v3 = strchr(s, 10);
  if ( v3 )
    *v3 = 0;
  return strlen(s);
}
```

图 4　sub_804858B 函数伪代码

可以看到该程序代码逻辑为先由 fgets 函数获取一个输入，检查其长度不为 0 后即通过 printf 函数输出。但是其通过 printf 函数输出时直接将输入内容作为 printf 的 format 参数，导致图 3 的第 13 行处存在一个格式化字符串漏洞。

为便于观察实验效果，本实验拟生成通过该格式化字符串漏洞将对 printf 函数的 got 表地址改写为 system 函数的地址的测试用例。经 FSVDTG 系统后分析得到分析结果如图 5 所示。发现 0xbffff53c 处的 format 参数被符号化。使用 ghex 查看生成的测试用例，其中的具体内容如图 6 所示。

```
8220 printf() entry-hook invoked:
8221 address = 0x8049974
8222 data = 0xf7e3cda0
8223 format = 0xbffff53c
8224 20 [State 0] Found Symbolic Array at 0xbfffe478, width 12
8225 20 [State 0] Found Symbolic Array at 0xbfffe488, width 4
8226 20 [State 0] Found Symbolic Array at 0xbffff53c, width 60
8227
8228 The format is symbol:
8229 LinuxPrintfAPI: address to w/r= 0x8049974
8230 format start at : 0xbffff53c
8231 formatArg_address = 0xbffff520
8232 Generating testcase for print-write...
8233 ----use %hn---
8234 20 [State 0] Pruned 162 out of 0 constraints
8235 20 [State 0] ---Write testcase to file /home/acdax/Desktop/test/testcase-1.bin---
8236 Generating testcase for print-write...
```

图 5　系统分析结果

```
00000000 74 99 04 08 76 99 04 08 25 35 32 36 33 32 78 25 t...v...%52632x%
00000010 37 24 68 6E 25 31 30 38 31 39 78 25 38 24 68 6E 7$hn%10819x%8$hn
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000030 00 00 00 00 00 00 00 00 00 00 00 00             ............
```

图 6　ghex 查看测试用例结果

```
$ ls
[DEBUG] Sent 0x3 bytes:
    'ls\n'
[DEBUG] Received 0x2b bytes:
    '1.py  pingme  test  test2  testcase-1.bin\n'
    '\n'
1.py  pingme  test  test2  testcase-1.bin

$ cp ./test ./test3
[DEBUG] Sent 0x12 bytes:
    'cp ./test ./test3\n'
[DEBUG] Received 0x1 bytes:
    '\n'

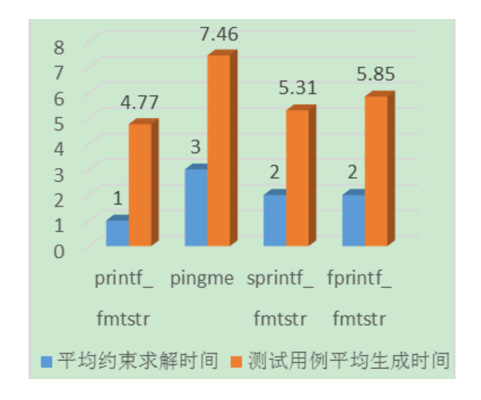
$ ls
[DEBUG] Sent 0x3 bytes:
    'ls\n'
[DEBUG] Received 0x32 bytes:
    '1.py  pingme  test  test2  test3  testcase-1.bin\n'
    '\n'
1.py  pingme  test  test2  test3  testcase-1.bin
```

图 7 以测试用例为输入后 pingme 运行结果

将本系统生成的测试用例作为输入后效果如图 7 所示。可见 ls 和 cp 命令执行均成功，因此测试用例执行后 printf 函数的 got 表地址被成功改写为 system 函数的地址，故所生成的测试用例有效。

本文选用了四个含缓冲区溢出漏洞的二进制实验程序进行实验，其中一个程序为典型的由 printf 使用不规范引发的格式化字符串漏洞；一个程序选自 NJCTF 2017 的 200 分 PWN 题 pingme，其也是由 printf 引起的；其余两个测试程序分别含有由 sprintf 和 fprintf 引起的格式化字符串漏洞。上述测试程序均采用 C 语言或 C++进行编写。实验宿主机配置为：Intel Core i7 CPU@3. 60 GHz、16 GB 内存、ubuntu 系统；虚拟机配置为 1 GB 内存， ubuntu 系统。记录 4 个程序在本系统中的平均约束求解时间和测试用例平均生成时间如表 6 所示。

表 6　FSVDTG 系统下实验结果/s



为与已有其他方法进行对比，对上述 4 个测试程序再分别采用直接运行和使用 CRAX、AFL、pwntools 的方式进行测试，记录其直接运行时间及各工具下从运行到测试用例生成时间，记录结果如表 7 所示。所记录时间均为平均挂钟时间。各项工具中 CRAX 是与本系统工作原理一致且都是基于 S2E 实现的，而 pwntools 虽然不是基于符号执行实现的，但却是自动生成格式化字符串漏洞测试用例的典型工具，AFL 是基于模糊测试生成测试用例的经典工具。

表 7　各工具测试用例平均生成时间结果/s

实验工具 / 测试程序	printf_ fmtstr	pingme	sprintf_ fmtstr	fprintf_ fmtstr
直接运行	0. 20	0. 34	0. 27	0. 28
FSVDTG	4. 77	7. 46	5. 31	5. 85
CRAX	N/A	N/A	N/A	N/A
pwntools	0. 31	0. 57	N/A	N/A
AFL	31	N/A	39	48

针对该实验结果分析如下：

CRAX 是较为典型的基于符号执行的测试用例自动生成工具，但其在漏洞建模方面仅对 IP 寄存器被劫持的情况进行测试用例生成，尽管其对多数控制流劫持类漏洞效果较好，但单纯的格式化字符串漏洞采用一般测试输入是并不能覆盖 IP 寄存器的，故 CRAX 对 4 个程序测试结果均为 N/A，即不适用，而本文所提出的方法也可作为对 CRAX 的一种补充。

pwntools 的执行效率较高，针对 printf 函数引起的格式化字符串漏洞生成测试用例速度快，但其需通过 printf 的打印值确定 format 参数与输入缓冲区间的偏移，故无法处理由 sprintf 和 fprintf 等函数引起的格式化字符串漏洞，但 pwntools 所生成的测试用例均可直接触发格式化字符串漏洞进行指定地址读写，而非单纯的崩溃。

AFL 是模糊测试的常用工具之一，其采用 Fuzzing 技术进行测试用例生成，其适用面广，但生成测试用例过程中盲目性、随机性较高，因此生成测试用例时间长。AFL 最终对其中 3 个程序成功生成了测试用例，对 pingme 程序未能生成测试用例，其原因主要是由于 pingme 程序中含有一个可进行持续输入的死循环，导致 AFL 无法跳出循环，最终未能生成测试用例。同时 AFL 生成的测试用例主要功能只是单纯触发崩溃而非进行指定地址读写。

实验结果表明，本文所实现的 FSVDTG 系统所生成的测试用例也可直接用于指定地址读写。其基于符号执行实现，由于需要对数据进行符号化处理等复杂操作，其处理速度相较于 pwntools 较慢，但可对多种函数引起的格式化字符串漏洞进行测试用例生成。

4　结束语

本文总结了格式化字符串漏洞的相关原理，并设计实现了一种基于符号执行的格式化字符串漏洞自动检测与测试用例生成的方法，以检测 linux 下二进制程序存在的格式化字符串漏洞，判定其是否会导致任意内存读写危害，并生成完成任意地址读写功能的稳定测试用例。最后通过实验验证了测试用例的有效性，并与同类方法进行了对比。

参考文献：

[1] 林桠泉, 漏洞战争 [M]. 北京: 电子工业出版社, 2016: 227. (Lin Yaquan, The war of vulnerability [M] , Beijing: Publishing House of Electronics Industry, 2016: 227.)

[2] King J C. Symbolic execution and program testing [J]. Communications of the ACM, 1976, 19 (7): 385-394.

[3] Moura L D, Bjørner N. Z3: An efficient SMT solver [J]. Lecture Notes in Computer Science, 2008, 4963: 337-340.

[4] Dutertre B, de Moura L. A fast linear-arithmeticsolver for DPLL (T) [C]// Proc of the 18th International Conference on Computer Aided Verification. Berlin: Springer, 2006: 81-94.

[5] Ganesh V, Dill D L. A decision procedure for bit-vectors and arrays [C]// Proc of International Conference on Computer Aided Verification, 2007: 519-531.

[6] ChIPounov V, Georgescu V, Zamfir C, *et al.* Selective symbolic execution [C]// Proc of Workshop on Hot Topics in System Dependability. 2009: 1286-1299.

[7] Huang S K, Huang M H, Huang P Y, *et al.* CRAX: software crash analysis for automatic exploit generation by modeling attacks as symbolic Continuations [C]// Proc of the 6th IEEE International Conference on Software Security and Reliability. [S. l.] : IEEE Computer Society, 2012: 78-87.

[8] Yan S, Wang R, Salls C, *et al.* SOK: (state of) the art of war: offensive techniques in binary analysis [C]// Proc of IEEE Security and Privacy. 2016: 138-157.

[9] 程铃. MANET 入侵检测技术的研究 [J]. 微电子学与计算机, 2010, 27 (6): 64-67. (Cheng Ling, Study on instrusion detection for mobile Ad hoc networks [J]. Microelectronics & Computer, 2010, 27 (6): 64-67.)

[10] TESO Security Group. Exploiting format string vulnerabilities [EB/OL] (2001) [2018-04-04]. http://www. tech-faq. com/format-string-vulnerability. Html.

[11] Wilander J, Kamkar M. A Comparison of publicly available tools for dynamic buffer overflow prevention [J]. Ndss, 2002, 2015 (5): 45-50.

[12] 王清, 0DAY 安全: 软件漏洞分析技术 [M]. 2nd ed. 北京: 电子工业出版社, 2011: 245. (Wang Qing, 0DAY security: software vulnerability analysis technology [M]. 2nd ed. Beijing: Publishing House of Electronics Industry, 2011: 245.)